Matthew

WebAssembly backend for the OCaml compiler

June, 2023

Contents

1 Introduction **6**
- 1.1 WebAssembly . . . 6
- 1.2 OCaml . . . 6
- 1.3 Motivation for porting to WebAssembly . . . 7
- 1.4 Related work . . . 8

2 Preparation **9**
- 2.1 Starting point . . . 9
- 2.2 Requirements analysis . . . 9
- 2.3 OCaml compiler . . . 9
  - 2.3.1 Front End . . . 10
  - 2.3.2 Middle End . . . 11
  - 2.3.3 Back End . . . 14
  - 2.3.4 Bootstrapping and build system . . . 17
- 2.4 WebAssembly . . . 18
  - 2.4.1 WAT . . . 18
  - 2.4.2 Control Flow . . . 19
- 2.5 Tools . . . 20
- 2.6 Project management . . . 20
- 2.7 Software engineering methodology . . . 20
- 2.8 Software licences . . . 20

3 Implementation **21**
- 3.1 Build system . . . 21
- 3.2 Architecture setup . . . 22
  - 3.2.1 arch.ml . . . 22
  - 3.2.2 CSE.ml . . . 23
  - 3.2.3 proc.ml . . . 24
  - 3.2.4 reload.ml . . . 24
  - 3.2.5 selection.ml . . . 24
  - 3.2.6 scheduling.ml . . . 24
  - 3.2.7 emit.mlp . . . 25
- 3.3 Code generation . . . 25
  - 3.3.1 Assembly format . . . 25
  - 3.3.2 Data . . . 25

3.3.3 Functions . . . 27
3.3.4 Linear instructions . . . 27
3.4 Memory management . . . 29
3.4.1 Allocation . . . 29
3.4.2 Garbage collection . . . 29
3.5 Control flow . . . 30
3.5.1 Implementation . . . 33
3.5.2 Exceptions . . . 34
3.6 Runtime . . . 35
3.7 Repository overview . . . 36

4 Evaluation **37**
4.1 Tests . . . 37
4.2 Performance . . . 39
4.3 Decision against upstreamimg . . . 42
4.4 Summary . . . 43

5 Conclusions **44**
5.1 Lessons Learned . . . 44
5.2 Future Work . . . 44

Bibliography **45**

List of Figures

2.1 OCaml compiler pipeline 10
2.2 Back end pipeline 14
2.3 Build process 17

3.1 DAG 31
3.2 DAG with blocks 32

4.1 Integer performance 39
4.2 Allocation performance 40
4.3 Out of bounds cost 41
4.4 Exception 42

Listings

2.1 Example parse tree 11
2.2 Example typed tree 11
2.3 example.ml 12
2.4 example.ml - Lambda IR 12
2.5 example.ml - CLambda IR 13
2.6 example.ml - Cmm 15
2.7 example.ml - after instruction selection 16

3.1 WAT 26
3.2 Emscripten format 26
3.3 WebAssembly text format comparison 26
3.4 Depth-first search 33
3.5 Back edges 34

Chapter 1

Introduction

This chapter will introduce the two main components of an OCaml to WebAssembly compiler. The programming language OCaml, and the portable binary format WebAssembly.

1.1 WebAssembly

WebAssembly is a portable binary code format, designed as a compilation target to enable efficient execution of high-level languages on the web. It is a stack-based language and typically JIT compiled.

As WebAssembly runs on the web, security was a key design principle. One such example is that WebAssembly programs are limited to only accessing a linear region of memory that is allocated for them. As such it is not possible for WebAssembly programs to access JavaScript objects or access arbitrary memory in the browser's process.

Another example is that all instructions and functions are typed, consuming and producing a fixed number of values on the stack. Compile-time checks are performed[1] to ensure instructions can never pop from an empty stack and functions return with only the return value specified by their type left on the stack.

WebAssembly is often used to port existing applications to the web. Languages such as C, C# and Rust have compilers that target WebAssembly. In addition, game engines such as Unity support WebAssembly as a target, allowing for running complex games in the browser. WebAssembly has also found its use in speeding up CPU-intensive tasks that would otherwise have to be written in JavaScript. One such example is speeding up cryptographic functions.

1.2 OCaml

OCaml is a functional programming language derived from Standard ML. Historically, it was developed as an extension to the Caml programming language with object-oriented features added. However, these are rarely used in practice[2].

[1] This will typically be done when producing the WebAssembly (WASM) binary and when it is loaded at run time, although only the latter is necessary for security.

[2] The compiler itself is a notable exception.

OCaml is used by several large companies such as Microsoft, Facebook, Bloomberg and most notably Jane Street where OCaml powers billions of dollars in daily transactions.

Some notable OCaml projects include:

Ocsigen server
Webserver, an alternative to software such as Apache or Nginx

Google Drive FUSE driver
FUSE[3] driver to enable access to Google Drive on Linux

MirageOS
Library for building unikernels[4]

Coq
Well-known theorem prover

Core
Alternative to OCaml's sparse built-in standard library.

1.3 Motivation for porting to WebAssembly

There are two key reasons for compiling OCaml to WebAssembly. The first and most obvious reason is to run OCaml in on the web. Js_of_ocaml is an existing compiler that allows for running OCaml on the web by compiling the OCaml bytecode to JavaScript. There are several reasons why compiling to WebAssembly may be a good alternative.

Performance Allocation in OCaml bytecode is compiled to allocation of JavaScript arrays by Js_of_ocaml. This means performance is limited by the allocation and garbage collection strategy of the JavaScript engine used. This is likely to incur a performance hit as functional languages make heavy use of linked lists and small, short-lived allocations, something that is not as commonly done when writing JavaScript. As such, most JavaScript engines' allocation is not optimised with this in mind. Additionally, functional patterns such as currying are optimised in OCaml's native compiler in a way that cannot be done when targeting JavaScript.

Compatibility Due to JavaScript not natively supporting 64-bit integers, Js_of_ocaml uses 32-bit integers, a significant downgrade over OCaml's standard, 63-bit[5] integers. This means any existing OCaml code relying on integers larger than the 32-bit limit needs to be modified to work correctly when compiled with Js_of_ocaml.

[3]Filesystem in Userspace (FUSE) is an interface for Unix-like kernels to enable running of filesystem drivers in userspace.

[4]A unikernel is a specialised operating system built to run a single application. As such they can be compiled together and run in the same address space.

[5]The reason behind 63-bit rather than 64-bit integers being used is explained later.

Right now, OCaml's native compiler only supports a limited number of architectures; at the time of writing that is AMD64, ARM64[6], PowerPC, RISC-V and s390x[7]. This means, outside of Js_of_ocaml, it is impossible to run OCaml on other architectures[8]. A WebAssembly compilation target would provide for a single executable that can run on any architecture.

1.4 Related work

As previously mentioned, the Js_of_ocaml compiler produces JavaScript from OCaml bytecode [12]. ReScript (formerly BuckleScript) is a fork of OCaml that targets JavaScript [2]. Unlike Js_of_ocaml, ReScript works from the Lambda[9] format used inside the OCaml compiler. It also has the goal of more tightly integrating with the JavaScript ecosystem, targeting npm rather than opam. There has been at least one other attempt to compile OCaml to WebAssembly [11]. This worked by converting the Cmm[9] representation used in the compiler directly to WebAssembly. This attempt went a long way towards being feature-complete, although it stopped short of implementing garbage collection.

[6] i386 and 32-bit ARM were supported in OCaml 4.14. However, they have since been dropped.

[7] Only AMD64 and ARM64 are supported in OCaml 5.0. PowerPC, RISC-V and s390x were all supported in 4.14 and there are plans for them to return in OCaml 5.1+.

[8] It would be possible to use some sort of binary translation layer or full system emulator like QEMU but this would incur a substantial performance overhead.

[9] Explained in section 2.3.

Chapter 2

Preparation

In order to successfully understand the work undertaken to complete the project. It is necessary to understand the basis for the project, as well as some key concepts of the OCaml compiler and WebAssembly.

2.1 Starting point

The basis for the code in the project is version 4.14.0 of the OCaml compiler, specifically, commit #83762af41d5a302ed793d6fb4bbe18a3447e18b1 of the OCaml repository on GitHub.

2.2 Requirements analysis

The success of the project was contingent on the ability to compile a subset of OCaml, specifically OCaml without exceptions or external function calls[1], to WebAssembly. A key part of this is supporting the OCaml runtime's garbage collector. As mentioned earlier, this is where other attempts to write a backend for WebAssembly have failed.

Two possible extensions for the project were presented. The first being to implement calling external functions. The second, to support OCaml exceptions.

If both extensions were carried out, then that would imply support for the entire OCaml language. A test of this would be compiling an existing OCaml project, such as the compiler, into WebAssembly.

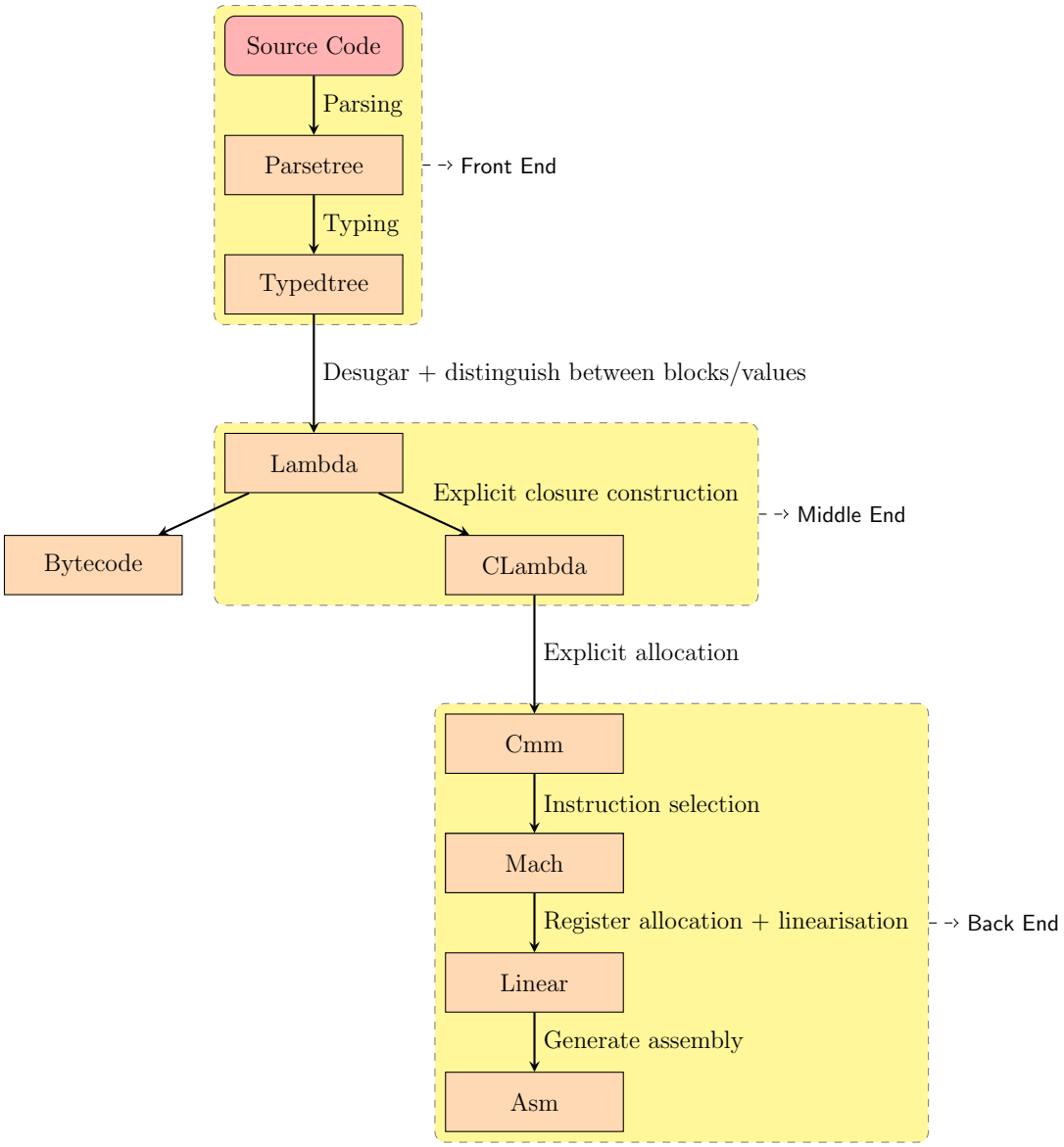
2.3 OCaml compiler

The OCaml compiler is a self-hosting[2] compiler. It can be configured to target bytecode or machine code for certain architectures, although this must be chosen at compile time. When producing native code, a C runtime is linked in. This performs startup and implements garbage collection. The runtime also provides APIs for OCaml's standard library, such as I/O functionality and string manipulation.

[1] The ability to call C code.

[2] Can compile itself.

Figure 2.1: OCaml compiler pipeline



The OCaml compiler consists of three stages: the front end, the middle end and the backend. The front end deals with parsing and type checking, the middle end performs desugarisation and optimisations such as inlining while the backend generates the assembly. An overview of the pipeline can be seen in Figure 2.1. We shall now take a detailed look at the different stages of the pipeline.

2.3.1 Front End

The first step the front end takes is to parse an OCaml source code file into a parse tree. At this stage all unnecessary detail in the source code such as comments and whitespace will be thrown away. For the following OCaml code:

```
let one = 1
```

Listing 2.1: Example parse tree

```
1  [
2    structure_item
3      Pstr_value Nonrec
4      [
5        <def>
6          pattern
7            Ppat_var "one"
8          expression
9            Pexp_constant PConst_int (1,None)
10     ]
11 ]
```

Listing 2.2: Example typed tree

```
1  [
2    structure_item
3      Tstr_value Nonrec
4      [
5        <def>
6          pattern
7            Tpat_var "one/4"
8          expression
9            Texp_constant Const_int 1
10     ]
11 ]
```

once parsed, the compiler will represent this internally as the parse tree seen in Listing 2.1. Next, the compiler will perform type checking and turn this into the typed tree seen in Listing 2.2. One key difference worth noting is on line 7, where the string 'one' is transformed into 'one/4'. The number after the slash is used to disambiguate multiple semantically different uses of an identifier within a single compilation unit.

2.3.2 Middle End

The job of the middle end is to transform the high-level parsed OCaml source into a lower-level intermediate representation (IR) consisting of basic operations. The first step in the middle end is to transform the typed tree into the Lambda IR format.

In order to understand the different stages of the compiler, we shall consider the following OCaml code, defining a function

```
lazy_inc_cons:int -> (unit -> int) list -> (unit -> int) list
```

Listing 2.3: example.ml

```
external ( + ) : int -> int -> int = "%addint"

let lazy_inc_cons x tl =
  match x with
  | -1 -> tl
  | x -> (fun () -> x + 1) :: tl
```

Listing 2.4: example.ml - Lambda IR

```
1 (seq
2   (let
3     (lazy_inc_cons/5 =
4       (function x/7[int] tl/8
5         (if (!= x/7 -1)
6           (makeblock 0 (function param/10[int] : int (+ x/7 1)) tl/8) tl/8)))
7     (setfield_ptr(root-init) 0 (global Example!) lazy_inc_cons/5))
8   0)
```

that conditionally appends a function to a list. To avoid linking in the standard library, thus simplifying the produced IR, we also define the addition function in the standard way (as a compiler intrinsic).

Lambda The Lambda IR format is the first IR format used in the OCaml compiler. At this stage, all high-level constructs such as pattern matching and modules will be desugarised. For example, pattern matching is turned into a combination of `if` and `switch` statements.

Another transformation performed is to create a distinction between blocks and values; values are objects such as integers and nullary constructors; blocks are heap-allocated objects such as lists and records.

Types are also erased in the conversion to the Lambda IR. By this point all type checking has been completed and any operation that needs type information, such as accessing the field of a record, has been transformed into a lookup from a block.

After the Lambda IR has been generated a simple optimisation pass is performed. One optimisation performed at this stage is to collapse aliased variables. This is the last stage for which the bytecode compiler and native compiler are the same.

As can be seen in line 5 of Listing 2.4, the Lambda IR generated for the OCaml code in Listing 2.3 desugars the `match` into a simple `if` statement. It can also be seen on line 6 that the lambda function is treated as a block whereas the integer defined within the function is not a block.

Listing 2.5: example.ml - CLambda IR

```
1  (seq
2    (let
3      (lazy_inc_cons/5
4        (closure
5          (fun camlTmp__lazy_inc_cons_5 2  x/7[int] tl/8
6            (if (!= x/7 -1)
7              (makeblock 0
8                (closure
9                  (fun camlTmp__fun_14:int 1  param/10[int] env/16
10                   (+ (field 2 env/16) 1))
11                 x/7)
12               tl/8)
13             tl/8)) ))
14     (setfield_ptr(root-init) 0 (read_symbol camlTmp) lazy_inc_cons/5))
15   0)
```

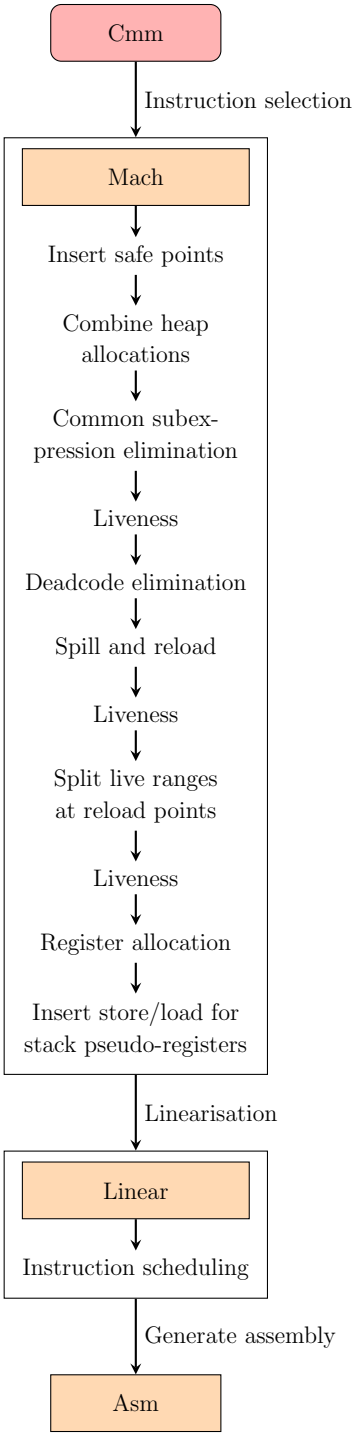
CLambda The next stage in the compiler is to transform the Lambda IR into the CLambda IR. The most significant change at this stage is to make closures explicit. This stage also contains a significant optimisation pass, the most notable optimisation being inlining.

Explicit closure construction can be seen on lines 4 and 8 in Listing 2.5. As the lambda function has a free variable, it must be captured. This can be seen at the end of the line 8 closure on line 11. Additionally, the lambda function obtains a new environment argument on line 9 with the variable `x` being accessed from it on line 10.

2.3.3 Back End

The job of the back end is to transform the code given to it by the middle end into assembly. To understand how this works, we shall now go through the different IRs used by the back end, showing how they work using our code sample from before. A detailed view of the backend can be seen in Figure 2.2.

Figure 2.2: Back end pipeline



Cmm The first IR used in the back end is the Cmm. Several changes occur at this stage, the first of which is that allocation is made explicit. This can be seen in line 3 of Listing 2.6, where the outer allocation defines the list node containing two values, the closure as its data and the function's parameter as its tail. The inner allocation constructs the closure, containing a pointer to the function, a magic value encoding information such as the arity of the function, and the closure variables. It can also be seen that, when allocating a block, a block header is included. For the list node, the value of the header is 2048, the upper 10 bits of which represent the size of the allocation, in this case 2. For the closure, the header is 3319, with the upper 10 bits representing a size of 3 and the lower bits, 247, representing a tag indicating that the block allocates a closure.

Another transformation at this stage is the lifting of all functions to the global scope. This can be seen on line 6, where the lambda function has been lifted out of the `lazy_inc_cons` function.

The final change to note is that the distinction between values and pointers to blocks is encoded using the least significant bit (LSB) of all integers. This is why integers on an n-bit architecture can only use n-1 bits, hence 63-bit integers on a 64-bit system. The purpose of this is so that the garbage collector (GC) can distinguish between values and pointers it needs to explore further. If the LSB is set, that indicates a value; if it is unset then it indicates a pointer. A consequence of this can be seen on line 7 where the number 2 is now added instead of 1 to preserve the LSB. This is equivalent to removing the marker, performing the addition and then re-adding it. If we assume that the input to the function already has its LSB set, which we notate

Listing 2.6: example.ml - Cmm

```
1 (function camlExample__lazy_inc_cons_5 (x/7: val tl/8: val)
2   (if (!= x/7 -1)
3     (alloc 2048 (alloc 3319 "camlExample__fun_14" 72057594037927941 x/7) tl/8)
4     tl/8))
5
6 (function camlExample__fun_14 (param/10: val env/16: val)
7   (+ (load val (+a env/16 16)) 2))
```

as

$$x.1 = (x << 1) \,|\, 1$$

then it follows that

$$\begin{aligned}((x.1 >> 1) + 1) << 1 \,|\, 1 &= (x + 1) << 1 \,|\, 1 \\ &= ((x << 1) + 2) \,|\, 1 \\ &= ((x << 1) \,|\, 1) + 2 \\ &= x.1 + 2\end{aligned}$$

It should be noted that this distinction is only required at certain points within a function such as allocation and function calls. This is because these are the places where the GC can be triggered.

Mach The next IR used in the compiler is the Mach format. Unlike the previous formats, this IR goes through many separate transformations[3] without changing format. As such it is possible to inspect each of the transformations individually. For brevity, only the final stage is included as an example.

The first transformation performed is instruction selection. This can be seen in Listing 2.7 where the code is now a list of sequential instructions. Once instruction selection has taken place, a series of optimisations such as common subexpression elimination and liveness analysis are performed. The final step taken at this stage is to perform register allocation. The example given in Listing 2.7 is taken from an x86-64 machine, meaning the registers shown are from that architecture.

Linear The final format used in the IR is the Linear format. The only change from the Mach format is that control flow has been flattened into jumps and labels. For example, line 4 in Listing 2.7 is replaced with

```
L101:
if x/29[%rax] ==s -1 goto L100
```

[3]As it is not important to understand these transformations in depth, only a few will be mentioned here. A complete list can be seen in Figure 2.2.

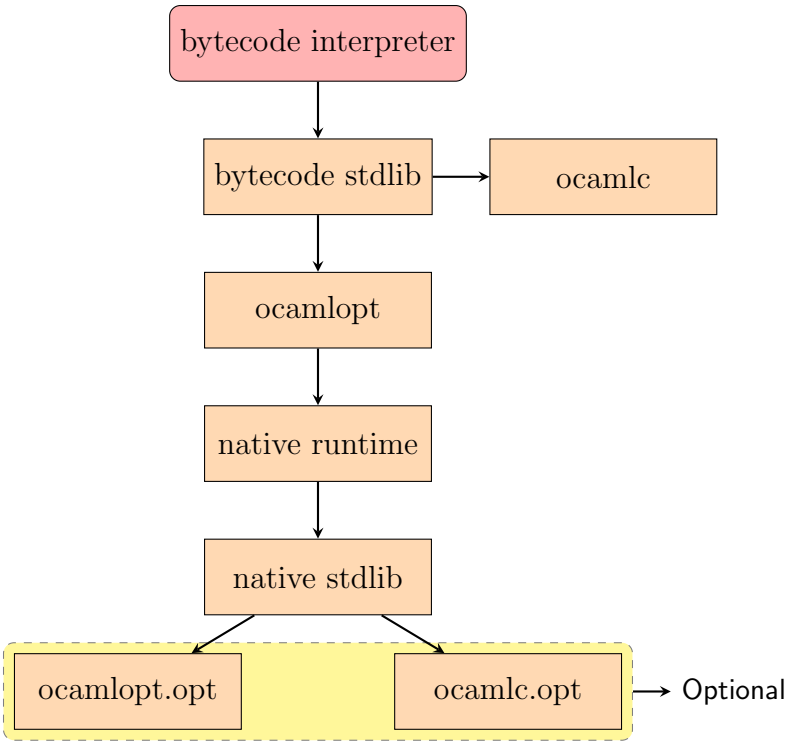
Listing 2.7: example.ml - after instruction selection

```
1  camlExample__lazy_inc_cons_5(R/0[%rax] R/1[%rbx])
2    x/29[%rax] := R/0[%rax]
3    tl/30[%rbx] := R/1[%rbx]
4    if x/29[%rax] !=s -1 then
5      V/31[%rdi] := alloc 56
6      V/31[%rdi] := V/31[%rdi] + 24
7      [V/31[%rdi] + -8] := 3319 (init)
8      I/32[%rsi] := "camlExample__fun_14"
9      val[V/31[%rdi]] := I/32[%rsi] (init)
10     I/33[%rsi] := 72057594037927941
11     val[V/31[%rdi] + 8] := I/33[%rsi] (init)
12     val[V/31[%rdi] + 16] := x/29[%rax] (init)
13     V/34[%rax] := V/31[%rdi] + -24
14     [V/34[%rax] + -8] := 2048 (init)
15     val[V/34[%rax]] := V/31[%rdi] (init)
16     val[V/34[%rax] + 8] := tl/30[%rbx] (init)
17     R/0[%rax] := V/34[%rax]
18     return R/0[%rax]
19   else
20     R/0[%rax] := tl/30[%rbx]
21     return R/0[%rax]
22   endif
23
24 camlExample__fun_14(R/0[%rax] R/1[%rbx])
25   env/30[%rbx] := R/1[%rbx]
26   V/31[%rax] := val[env/30[%rbx] + 16]
27   I/32[%rax] := V/31[%rax]
28   I/32[%rax] := I/32[%rax] + 2
29   R/0[%rax] := I/32[%rax]
30   return R/0[%rax]
```

with the else on line 19 being replaced with the label L100.

After linearisation, instruction scheduling is performed, allowing for architecture-specific reordering of instructions. Once this has been done, the Linear code is handed off to architecture-specific code for generating ASM.

Figure 2.3: Build process



2.3.4 Bootstrapping and build system

The OCaml compiler is built using a Makefile. The compiler does not support cross compilation. As such it must be built on the architecture it is intended to target.

The compiler can be built without an OCaml compiler installed on the system. A well-informed reader may ask how this is possible as the OCaml compiler is written in OCaml. To answer that, we must consider the entire build process of the OCaml compiler. An overview of the process can be seen in Figure 2.3

The first step in building the compiler is to build the OCaml bytecode interpreter. This is written in C therefore it does not pose a problem. Once this has been built it is possible to run the bootstrapping bytecode compiler which is shipped with the source code.

This bytecode compiler is then used to build the standard library, thus allowing for OCaml code to be compiled into bytecode. It is now possible to build either the native compiler or a new revision of the bytecode compiler. The next step in building the native compiler is to use the bytecode compiler to compile a native targeting compiler.

In order to use this new native compiler, two things must be compiled. The first is the native runtime. This is largely written in C, although a few stubs are written in ASM. Second, the standard library must be compiled into native code.

Once this has been done, a native compiler (ocamlopt) has been produced. Optionally, it is then possible to compile a native build of the bytecode (ocamlc.opt) and native (ocamlopt.opt) compilers.

2.4 WebAssembly

WebAssembly is a stack-based language. This means that, for example, `i64.add` pops two integers off the stack, adds them together and stores the result back on the stack. Another example is `i64.const 1337`, which pushes the constant 1337 onto the stack. WebAssembly does not have arbitrary stack access, meaning all operations can only read from and write to the top of the stack. As this would severely limit what is possible, WebAssembly also provides local storage in the form of locals.

An example of this is, `local.set 2`, which stores the top of the stack into the 3rd local[4]. Conversely, `local.get 2` would get the value of the 3rd local and push it onto the stack. Locals are function-scoped, meaning a callee cannot access its caller's locals. While the number of locals a function may access is not limited, it must declare at compile time how many it wishes to use. It must also say, for each local, what the type of that local is (e.g. `i64`, `i32`, `f64`).

WebAssembly also provides access to a linear region of memory. Much like traditional memory, each byte is assigned a sequential address, starting at 0. This allows for long-term storage of data that can be referenced using pointers. An example of this is, `i64.store 8`, which pops two integers off the stack. The first is a pointer to a location in memory. The second is the integer to be stored. The integer is then stored at an offset of 8 bytes from the address provided.

2.4.1 WAT

WAT is the canonical text-based representation of WebAssembly. It is a tree-based s-expression format in the style of Lisp. For example, the following code defines a WebAssembly program with a single function, `add`, which adds two numbers together and returns the result.

```
(module
 (type $i32_i32_=>_i32 (func (param i32 i32) (result i32)))
 (func $add (param $0 i32) (param $1 i32) (result i32)
  (local $2 i32)
  (local.set $2
   (i32.add
    (local.get $1)
    (local.get $0)
   )
  )
  (return
   (local.get $2)
  )
 )
)
```

[4]Locals are 0-indexed

As can be seen on line 3, the function takes in two integers and returns a single integer. On line 4, we can see that it defines a single local. This local is used on line 5 to store the result of the addition. This local is included for demonstration purposes and could be entirely optimised out.

2.4.2 Control Flow

WebAssembly is limited to structured control flow only. This is unusual for assembly languages, which tend to prefer arbitrary control flow. This means, rather than having labels and jump statements, WebAssembly is made up of structured blocks. These blocks are nested and can be broken out of with conditional or unconditional breaks. WebAssembly also has an `if` block which is simply sugar syntax over a regular block with a conditional break. This is there to help reduce the size of the generated code.

An example of control flow can be seen in the following example, defining a function `is_zero` that returns 1 `if` the input is 0 and 0 otherwise.

```
1  (module
2   (type $i32=>_i32 (func (param i32) (result i32)))
3   (func $is_zero (param $0 i32) (result i32)
4    (block $0
5     (br_if $0
6      (i32.eqz
7       (local.get $0)
8      )
9     )
10    (return
11     (i32.const 0)
12    )
13   )
14   (return
15    (i32.const 1)
16   )
17  )
18 )
```

It can be seen on line 4 that the function defines a block that returns 0. If, however, the input to the function is 0, then the check on line 6 will be true and as such line 5 will cause a branch out of the block. If the branch is taken, the function will then run the code to return 1.

WebAssembly also has a loop block. This is equivalent to the regular block except branching takes you to the beginning of the block, rather than out of it.

2.5 Tools

To compile the C portions of the compiler to WebAssembly, the Emscripten compiler was used. The work was completed on an Arch Linux system. GNU/Linux was a defacto requirement as the OCaml compiler and its build system are significantly harder to work with on Windows. Windows would also likely add additional bootstrapping problems. Development was done in Emacs using the Spacemacs configuration.

2.6 Project management

Throughout the project, Git was used for version control. This was chosen for several reasons. The first is that it allowed for easy backup by publishing the repository to GitHub. The second is that it enabled access to old versions of the code. This facilitated debugging after a bad change.

2.7 Software engineering methodology

In the development of this project, the evolutionary model was used. This allowed for rapid development of the code as and when it was needed, thus alleviating the need to plan everything upfront and instead allowing for plans to form as development took place. This was beneficial as it reduced the amount of research needed upfront and allowed for researching parts of the compiler when needed rather than risking running out of time to write the project due to spending all the time planning and researching. The evolutionary model also allowed for quick bug finding and fixing.

2.8 Software licences

The OCaml compiler is licensed under LGPL 2.1. This licence allows for use and redistribution of the source code. The only restriction is that the modified source code must be provided along with the compiled compiler. The licence does not place any restrictions on software built using the compiler. Emscripten is licensed under the permissive MIT licence. This licence places no restrictions on use of the Emscripten or its source code.

Chapter 3

Implementation

In this section, the implementation of the WebAssembly backend will be covered. This starts with the build system, which must be set up before any work can begin. It then moves on to setting up the architecture within the compiler. The majority of this chapter will cover the generation of WebAssembly code and the problems that arise with control flow. Finally, setting up the runtime will be discussed.

3.1 Build system

The first step in building a new backend for the OCaml compiler is to get the bytecode interpreter running on the architecture you wish to target. Luckily, the bytecode interpreter does not require any custom configuration to build, thus allowing for any Unix-like system with a C compiler and autoconf/make support to build the interpreter through the standard './configure' followed by 'make' commands. This still initially poses a problem as WebAssembly is not a Unix system. Fortunately, we can turn to the Emscripten toolchain. This provides a GCC-like toolchain for WebAssembly, allowing us to use the existing build system to build the interpreter. Unfortunately, Emscripten comes with some downsides, the main one being that it does not target pure WebAssembly[1]. This is due to the fact that the runtime environment provided by Emscripten is written in JavaScript.

In order to be able to configure the compiler for WebAssembly, we must update the 'configure.ac' file with knowledge of Emscripten. This is fairly simple and just involves adding

```
[wasm64-*-emscripten],
  [arch=wasm64; system=emscripten]
```

to the `AS_CASE([$host],...)` statement starting on line 1057. This lets the build system know what architecture we are targeting, something that is needed for architecture-specific code to be built.

As we can now build the bytecode interpreter, the next step is to run the interpreter with the bootstrap compiler to build the standard library. This is a problem as the build

[1]It is possible to tell it to target the WebAssembly System Interface (WASI) API standard but WASI does not yet have sufficient functionality for our purposes.

system assumes it can run the interpreter as a standard executable by invoking it from a shell e.g. './ocamlrun'. However, it is not possible to directly execute a WebAssembly program in this way. Luckily, Emscripten produces a JavaScript wrapper containing the runtime. This wrapper can be executed by passing it as an argument to Node.js. As such, the build system can be modified to use Node.js to run generated executables. This can be done by modifying the Makefile from

```
OCAMLRUN ?= $(ROOTDIR)/boot/ocamlrun$(EXE)
```

to

```
OCAMLRUN ?= node --experimental-wasm-memory64
↪  --experimental-wasm-return_call $(ROOTDIR)/boot/ocamlrun$(EXE)
```

The experimental features enabled by the arguments will be explained later.

After making the above change, it is then possible to run the bootstrapping compiler and compile the standard library.

3.2 Architecture setup

The implementation of a specific architecture in the OCaml compiler can be found in `asmcomp/<arch>`. Each architecture is required to implement the following 7 files:

- `arch.ml`
- `CSE.ml`
- `emit.mlp`
- `proc.ml`
- `reload.ml`
- `scheduling.ml`
- `selection.ml`

3.2.1 arch.ml

The file `arch.ml` contains basic configuration options to set up a given architecture. These options can be characterised into several groups.

The first of these groups is setting the endianness and word size. For example, to support WebAssembly, we define the following:

```
let big_endian = false

let size_addr = 8
let size_int = 8
let size_float = 8
```

as WebAssembly is little-endian and supports 64-bit ints and floats. More interesting is the pointer size, which we also set to be 64 bits. To do this we must enable the Memory64 extension for WebAssembly. This is because the current WebAssembly specification only supports 32-bit pointers. The Memory64 extension enables 64-bit pointers. However, it is currently a stage 3 proposal, so it must be enabled manually.

As Memory64 is experimental, a reader might question why it was chosen to be enabled. To answer that, consider the following OCaml code:

```
let singleton x = [x]
```

As `singleton` is polymorphic over all inputs, `x` could be a heap-allocated object. Since `singleton` can trigger the GC, it must assume that `x` is a pointer. This would cause a problem if pointers were only 32 bits and an integer greater than the 32-bit limit were passed into the function. It would, alternatively, be possible to implement addresses as 64-bit integers and truncate them on use. However, this has the potential to increase code size substantially.

The next batch of options regard addressing modes. These options exist to provide support for the complex addressing modes of an architecture like x86. As WebAssembly is quite a simple language, we shall implement only one addressing mode, indexed addressing. This will provide addressing as a simple integer offset from a base. We choose this because it is easy to map onto the `i64.load <offset>` WebAssembly instruction and because identity- and offset-based addressing are required by the OCaml compiler.

The final set of options are to specify custom, architecture-specific instructions. As WebAssembly is a simple language, we need not specify any custom instructions. We also need to specify for these instructions if they have side effects and if they can raise exceptions. As we are not adding any custom instructions we specify that they are always pure and never raise.

3.2.2 CSE.ml

The purpose of the `CSE.ml` file is to configure the common subexpression elimination optimisations to support architecture-dependent operations. It can also be used to override defaults for built-in operations if they map onto architecture operations with specific characteristics that may require special handling when performing CSE. As we do not implement any custom instructions, we implement this file as a minimal stub, simply using the defaults for everything.

```
class cse = object (_self)

inherit cse_generic as super

method! class_of_operation op =
  match op with
  | Ispecific(_) -> assert false
  | _ -> super#class_of_operation op

end

let fundecl f =
  (new cse)#fundecl f
```

3.2.3 proc.ml

In `proc.ml`, we are able to specify the calling convention and registers available to our architecture. As WebAssembly does not have registers in the traditional sense, this is the first of many mismatches in the memory model of the OCaml compiler and that of WebAssembly.

To resolve this, we create a series of fake integer and floating point registers available to the OCaml backend. We will later map them to WebAssembly locals. This will limit us to a finite number of locals, determined at time of compilation of the compiler, causing a fallback to the stack should it be exceeded. By setting this number of virtual registers to be sufficiently large, we can avoid this problem in practice.

We have the additional problem of not having an indexed stack in WebAssembly, something that the compiler assumes. To get around this, we shall create a fake stack, often called a shadow stack [6], using the linear memory provided to us by WebAssembly. Unless otherwise indicated, this is the stack that is being referred to.

We shall define the calling convention to be storing the arguments in the first suitable register (the first integer goes in the first virtual integer register). This is because there is no clean way to map OCaml's register-based memory model to the stack-based calling of WebAssembly.

3.2.4 reload.ml

The `reload.ml` file contains overrides for whether an operation's arguments can reside in memory, or if they must be in registers. It is unclear if providing a full implementation here would be worthwhile for WebAssembly, so we provide a minimal stub using the default of requiring that most operations use registers, leaving open the possibility of revisiting this if needed.

3.2.5 selection.ml

In `selection.ml`, we are able to customise the instruction selection process. There are two main reasons to do this. The first is to determine when to use architecture-specific instructions. As we have none, there is nothing to do here on that account. The second is to select the addressing mode. As we only have a single addressing mode, we default to using it in all cases. We are also able to specify if instructions support immediate values. Due to the stack-based nature of WebAssembly, we shall treat immediates as pushing a constant to the stack before our operation, allowing us to avoid placing constants in locals beforehand.

3.2.6 scheduling.ml

The `scheduling.ml` file allows us to provide custom options for the scheduling of instructions. As WebAssembly is JIT compiled, we again provide a minimal implementation since the JIT compiler will likely do a much better job than any heuristic implemented here.

3.2.7 emit.mlp

The generation of the WebAssembly itself goes in `emit.mlp`. To make the code simpler, the `mlp` file format is used. This is a preprocessed version of the standard `ml` format. For example, the following code:

```
let emit_const i =
  `constant_int {emit_int i}\n`
```

is processed into an `ml` file writing `constant_int <i>\n` to a file. The generated `ml` file for this example can be seen in the following code:

```
let emit_const i =
  (emit_string "constant_int "; emit_int i; emit_char '\n')
```

3.3 Code generation

Generation of assembly code in `emit.mlp` is split into four parts:

```
val fundecl: Linear.fundecl -> unit
val data: Cmm.data_item list -> unit
val begin_assembly: unit -> unit
val end_assembly: unit -> unit
```

producing the code for functions and data as well as the start and end of a file respectively.

3.3.1 Assembly format

In generating the assembly, there are several choices for the format to generate the WebAssembly in. The most obvious choice is the standard WAT format. It was decided against using this as the backend makes extensive use of symbol referencing, something that the WAT format doesn't support. It would be possible to resolve the symbols manually into offsets of the linear memory. However, that would substantially increase the work needed to complete the project. Instead, it was decided that the internal format used by Emscripten would be used. This format is designed to mimic more traditional assembly language text formats with the ability to reference symbols and define data using directives such as `.int64 0`. A comparison between the two formats can be seen in Figure 3.3. The comparison depicts a simple function that returns the previous value it was given, starting at 0.

3.3.2 Data

Generation of the data segment is fairly simple. For example, the code to generate integer data types is as follows:

```
(module
 (type $i32_=>_i32 (func (param i32)
 ↪  (result i32)))
 (import "env" "__linear_memory"
 ↪  (memory $mimport$0 1))
 (data (i32.const 0) "\00\00\00\00")
 (func $last (param $0 i32) (result
 ↪  i32)
  (local $1 i32)
  (local.set $1
   (i32.load
    (i32.const 0)
   )
  )
  (i32.store
   (i32.const 0)
   (local.get $0)
  )
  (local.get $1)
 )
)
```

Listing 3.1: WAT

```
  .text
  .functype        last (i32) -> (i32)
  .section        .text.last,"",@
  .globl         last
  .type         last,@function
last:
  .functype        last (i32) -> (i32)
  .local          i32
  i32.const        0
  i32.load        last
  local.set        1
  i32.const        0
  local.get        0
  i32.store        last
  local.get        1
  end_function
  .type         last,@object
  .section        .bss.last,"",@
  .globl         last
last:
  .int32         0
  .size         last, 4
```

Listing 3.2: Emscripten format

Listing 3.3: WebAssembly text format comparison

```
| Cint8 n ->
    `.byte         {emit_int n}\n`
| Cint16 n ->
    `.short         {emit_int n}\n`
| Cint32 n ->
    `.long         {emit_nativeint n}\n`
| Cint n ->
    `.quad         {emit_nativeint n}\n`
```

The only complexity in emission of data segments is when emitting symbols. This is because we must specify whether the symbol refers to a function or another data element. This information is needed by the linker due to the Harvard architecture of WebAssembly. Luckily, function symbols are only emitted in closures. We can work this out as the code to emit data is called for each distinct data item, meaning we can simply check if the first thing in this is either a symbol or an integer with the correct bit pattern. If one of these is the case, whenever emitting a symbol reference, we also emit `.type <symbol>, @function\n`, telling the assembler that the reference is to a function.

3.3.3 Functions

In order to compile a WebAssembly function, we must provide the correct type information. The first thing that is needed is the function signature. The second is the number and types of the locals used. The list of virtual registers that a function's arguments reside in is thrown away during linearisation. However, by making a small change, we can preserve this information into the assembly generation stage. We can then iterate over this list to obtain the correct signature for a function.

To generate the type information for locals, we must first find the set of all virtual registers used in our function. Fortunately, each instruction has a property `arg` and `res` containing all the inputs and outputs to the instruction. We can therefore simply iterate over each instruction in the function and collect the total set registers used. Once we have this we then order the virtual registers

$$r_0, \ldots, r_{n-1}$$

such that the first i registers equate to the i arguments

$$a_0, \ldots, a_{i-1}$$

where

$$r_k = a_k$$

We then create a bijection from the registers to the first n natural numbers where

$$r_k \mapsto k$$

We do this to allow us to treat the registers as locals when emitting instructions that write to and from the virtual registers. We now use this ordering of the registers to emit the local definitions in the form of a list of types. For example, a function taking in two integer arguments, adding them together, then converting the result into a float, using a temporary register to store the result of the conversion, would have function signature `(i64, i64) -> (f64)` and local type of `i64,i64,f64`. The calculated bijection would map the first argument to local 0, the second to local 1 and the temporary register to local 2.

3.3.4 Linear instructions

Function Calls

Implementing direct function calls is fairly simple. All that needs to be done is to load the arguments onto the WebAssembly stack and then to emit an instruction to call the function. Finally, an instruction to load the return value from the WebAssembly stack into the resultant virtual register is generated. As we are provided with a list of the registers the arguments to the function are stored in, we simply need to iterate over the list and use the bijection between registers and locals to load the locals into the WebAssembly stack.

Indirect calls, however, pose a problem. This is because all indirect calls pass a closure to the function. This causes a problem when calling functions that do not rely on a closure, such as functions without any free variables. A closure must always be passed in indirect calls as there is no way to know if we are calling such a function. In a traditional assembly language, this does not pose a problem as functions do not depend on the values of non-argument registers. However, since WebAssembly functions are typed, we cannot simply call a function with an additional argument. To solve this, we wrap every function not relying on a closure with a function that simply discards the last argument passed to it (as the environment is always passed last) and then calls the original function. We then ensure that any indirect references to this function, such as closures, refer to the wrapper rather than the original function[2].

We can then implement indirect calls, in the same way as regular function calls. The only two differences being that we also need to add the pointer to the function onto the WebAssembly stack and we need to calculate the type of the function, something that we can simply do from the arguments passed to it.

Immediate and indirect tail calls are implemented in the same way as their non-tail counterparts but using the `return_call` and `return_call_indirect` instructions.

Basic Operations

The implementation of basic operations is quite simple. For example, integer operations are implemented by the following code

```
| Wop(Iintop op) ->
    (fun () ->
    emit_get_reg i.arg.(0);
    emit_get_reg i.arg.(1);
    let instr = name_for_int_operation op in
    `i64.{emit_string instr}\n`) |>
    emit_set_reg i.res.(0)
```

that loads the two arguments onto the WebAssembly stack, then converts the IR integer operation into a WebAssembly instruction and emits it. Finally, an instruction to store the result from the WebAssembly stack into a local is generated.

Another example is the implementation of `Iload`.

```
| Wop(Iload(chunk, Iindexed ofs, _mut)) ->
    let dst = i.res.(0) in
    let src = i.arg.(0) in
    emit_get_reg src;
    `i64.const {emit_int ofs}\n`;
    `i64.add\n`;
    let instr =
      match chunk with
```

[2]We technically do this in reverse, by having the default be the wrapped function and calling the unwrapped version in direct calls.

```
    | Word_int ->
        "i64.load"
    | Word_val ->
        "i64.load"
    | Byte_unsigned ->
        "i64.load8_u"
    | ... -> ...
in
`{emit_string instr} 0\n`;
emit_set_reg dst (fun _ -> ())
```

This works by first pushing the address to load from onto the WebAssembly stack, then generating the correct instruction to load the value from memory (8-bit vs 64-bit vs float etc). Finally, an instruction to store this in the virtual register is generated.

The majority of instructions can be implemented in a similar way to the examples given and are therefore not included here.

3.4 Memory management

OCaml maintains a minor (young) and major heap. All allocations in OCaml code are done on the young heap. Memory on the major heap can only be allocated and freed from C code. When the GC is triggered, it performs a compactification whereby everything live on the minor heap is moved to the major heap. The young heap is implemented as a linear block of memory with a pointer (`caml_young_ptr`) to the next free block. When allocating, the pointer is simply moved by the size of the requested allocation (in multiples of the block size). This continues up until the young heap is full, at which point the GC is called.

3.4.1 Allocation

Typically, an OCaml backend picks two registers to use as global values. The first of these contains `caml_young_ptr`. The second contains a pointer to a global state (`caml_state`). This state itself contains a pointer to the end of the young heap. Every time an allocation is needed, it is first checked if the allocation would overflow the young heap. If it would then the GC is invoked.

As WebAssembly does not have registers and locals are unique to a function, we cannot do this. Instead, we choose to create two WebAssembly globals to contain `caml_young_ptr` and `caml_state`. We can then use these two globals to implement allocation in the standard way.

3.4.2 Garbage collection

OCaml's garbage collection system is largely written in C. The only thing a backend needs to do is assist in the marking of the roots. Typically, when a function call occurs,

all live registers are pushed to the stack. This information is then stored at compile time for every call site. This information is stored in a frame table that is created for each assembly file. This table contains a list of structures. Each structure contains the return address of each call site, as well as a list containing the offset into the current stack frame for each live variable pointer. The structure also contains the size of the stack frame.

When the garbage collector is called, the return address into the OCaml code is looked up to access the frame information. This is then used to mark the roots located in the stack frame. As the size of the stack frame is also stored, the entire stack can be walked, forming a linked list of frames to scan for roots. Using these roots, garbage collection is performed, replacing any moved address on the stack with the new value.

Implementing this in WebAssembly poses two problems, both due to the non-addressable nature of the stack. The first is that we cannot simply push the virtual registers onto the WebAssembly stack, as there would be no way for a function later in the call chain to access them. The second is that we do not have return addresses.

We solve both of these problems by creating a virtual stack located in WebAssembly's linear memory. For this, we use the same virtual stack that is used in case we exhaust all of our virtual registers. As the backend assumes all registers are destroyed at a call site, the IR code to store the registers on the stack will be generated at the spill and reload stage seen in Figure 2.2. For every instruction, we are provided with a set of live variables. At function calls, all of these will reside on the stack. We can then use this to construct the frame table required to tell the runtime where in our stack the live variables are located, thus solving the first problem.

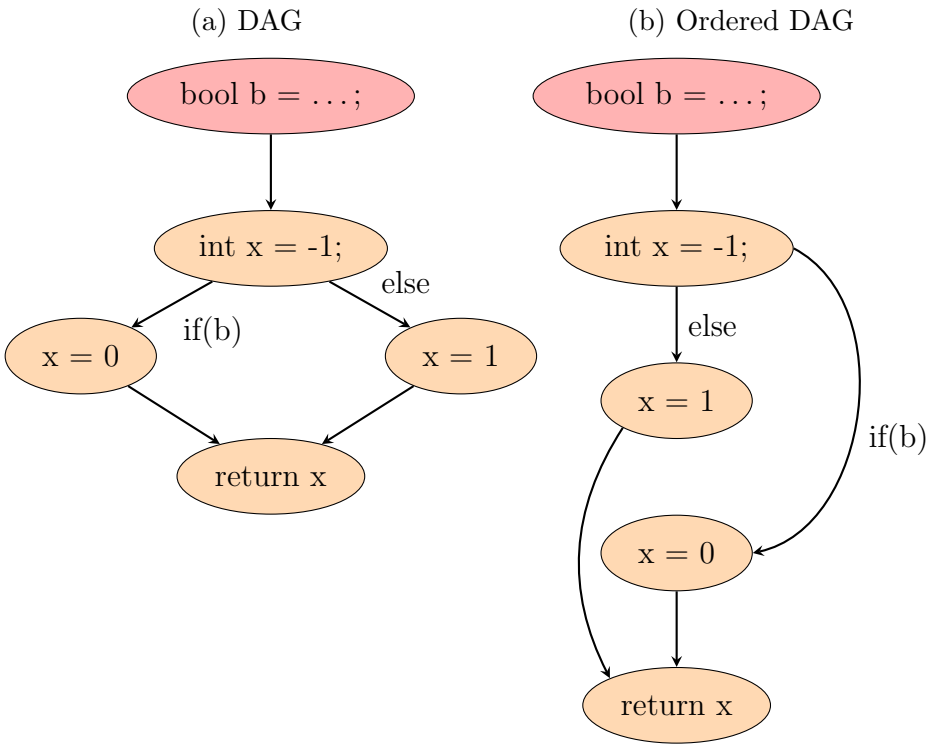
The second problem is a bit tricker as there is no way to obtain a return address in WebAssembly. Fortunately, the value of the return address is not important. Only that each call site has a unique integer associated with it. While within a file, we could just assign each call site a sequential integer, this would not guarantee uniqueness across files. As we compile each file separately, there is no way to construct some globally unique integer directly. Instead, for each call site, we emit into the data segment the byte `0`. We then use the address of this byte as our globally unique value. This works as, when linked together, memory addresses are guaranteed to be unique. We then store this address in the frame table and in the stack at call sites.

3.5 Control flow

The control flow used in OCaml's backend is unstructured control flow. This takes the form of a linear list of instructions with labels and branches. This can also be thought of as a control flow graph (CFG) of basic blocks. As WebAssembly does not have unstructured control flow support, we must find a way to turn our unstructured control flow into structured control flow.

By a folk theorem [4] often attributed to the Böhm-Jacopini [1] proof, it is possible to do this by first labelling each basic block with a number, then creating a variable containing the index of the starting block. Afterwards, a switch statement contained within a loop is appended, mapping the label to each basic block. At the end of each

Figure 3.1



basic block, we then perform our control flow by simply updating the variable to the index of the block we wish to branch to.

This method, although criticised by Donald Knuth [8], is guaranteed to work for all possible control flow. However, it has a few serious downsides. The first is that for reasons that will become apparent later, it is not obvious how exceptions would be implemented this way. The second is that this will be optimised poorly by the JIT compiler executing the WebAssembly. In turn, this will mean the performance of the generated assembly will likely not be acceptable.

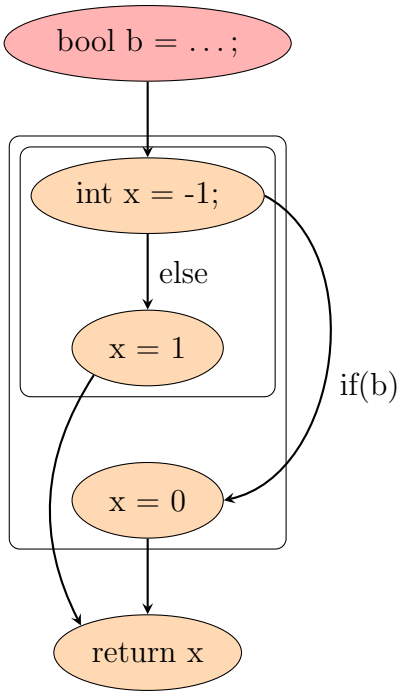
Sadly, it is not, in general, possible, without introducing an additional variable or node splitting, to turn arbitrary unstructured control flow into structured control flow [9]. It is, however, possible to turn reducible control flow into structured control flow. Reducible control flow is any CFG where loops can only be entered into for the first time in a single place [5]. More formally, this is a graph where it is possible to partition the edges into forward and back edges such that:

- Forward edges form a directed acyclic graph (DAG).
- For every back edge (A, B) B dominates A.

B can be said to dominate A if all paths from the starting node to A go through B. Fortunately, OCaml's backend only produces control flow in this form. If it were to produce arbitrary control flow, this would require either falling back to the loop method or node splitting.

We first consider how to structure non-looping control flow. That is to say, the case where the CFG is a DAG. In Figure 3.1(a), a simple control flow for a program that either

Figure 3.2: DAG with blocks



returns 0 or 1 can be seen. The first step in turning the DAG into structured control flow is to order the DAG in topological order. A topological order for a DAG is an ordering such that, for every edge (A, B) in the graph, A comes before B [3]. The ordered DAG can be seen in Figure 3.1(b).

Once we have put the DAG into topological order, we can then perform the stackifier algorithm [10, 7]. To do this, we loop through each edge (A,B) in the graph. We then construct a block such that the last node in the block is the predecessor of B. The first node in the block is either A or, if A is already in a block, the first node of the outermost block containing A that does not contain B. For our example from Figure 3.1, we can see the blocks that would be constructed in Figure 3.2.

Constructing this into pseudocode gives us:

```
1  bool b = ...;
2  block 1
3    block 0
4      int x = -1;
5      if (b) break 0;
6      x = 1;
7     break 1;
8    end
9    x = 0;
10 end
11 return x;
```

As can be seen on line 5, if the condition is true, we break out to the end of the block, meaning line 9 gets executed. If the condition is false, we continue to execute line 6 and

```
let dfs cfg f init =
  let visited = Hashtbl.create 10 in
  let rec visit src f acc =
    if not (Hashtbl.mem visited src) then begin
      Hashtbl.add visited src ();
      List.iter (fun dst ->
          visit dst f (f src dst acc)
       ) (get_adjacent_nodes cfg src);
    end
  in
  visit cfg.entry f init
```

Listing 3.4: Depth-first search

then break out to line 11.

To implement loops, we first separate out our CFG into forwards and back edges as described earlier. We then use the DAG that results from the forward edges to perform a topological sort. Before creating the blocks, we must first construct the loops. To do this, we do the reverse of what we did for blocks. This means that if we have a loop with entry node B and an edge (A,B) into the loop, the loop must start at node B and end with either B or the outermost loop containing B but not A. We then perform block insertion as before.

3.5.1 Implementation

To implement this in the OCaml compiler, we must first turn the linear instructions we are given into a CFG. This is trivial to do by iterating through the instructions and turning linear instructions under labels to nodes and branch instructions to edges. As branches are not required to appear at the end of a label, we must sometimes split up labels into multiple nodes.

Once we have our CFG, we need to calculate our forward and back edges. To do this, we perform a depth-first search on our CFG. The code in Listing 3.4 works by recursively descending down each edge of the CFG. A hash table is created, storing each visited node to prevent double visitation. The `dfs` function also takes in a function and an accumulator. The function is called every time a node is visited. The accumulator is threaded along the path to a certain node. It is important that this accumulator is only threaded through a node's descendants and discarded once having left the node. This is because CFGs are directed.

We then call `dfs` with a function that keeps track of all nodes visited along the path. This can be seen in Listing 3.5. If we ever find such a node, we add it to the list of back edges. Once we have our back edges, we then construct a DAG by removing the back edges from the CFG. We then implement the algorithm described above to place the blocks and loops. Once this is done, we flatten the blocks and loops into a sequence of instructions that can then be emitted as usual.

```
let back_edges = ref [] in
let f src dst visited =
  match LblSet.mem dst visited with
  | true ->
      back_edges := (src, dst) :: !back_edges;
      visited
  | false ->
      begin if src = dst then
        back_edges := (src, dst) :: !back_edges
      end;
      LblSet.add src visited
in
dfs cfg f LblSet.empty;
```

Listing 3.5: Back edges

3.5.2 Exceptions

Exceptions pose a challenge to implement in WebAssembly. This is because WebAssembly does not have any support for non-local control flow. Compilers wishing to support non-local control flow in WebAssembly, such as to implement C++ exceptions, have resorted to calling out to JavaScript functions and using the built-in JavaScript exception support. This, however, is very slow. Instead, we will use the WebAssembly exceptions proposal. This provides us with a structured try-catch-like exception handling. This can be seen in the following code:

```
try <type>
  # code
catch_all
  # catch code
end
```

Unfortunately, exceptions are unstructured in the Linear IR. This is because the back-end expects us to implement exceptions by storing a pointer to the exception handler in a register and jump to it when we raise an exception.

To understand how the Linear IR implements exceptions, consider the following code:

```
1 push trap L109
2 ... # code to catch
3 pop trap
4 goto L108
5 L109:
6 enter trap
7 ... # handle exception
8 L108:
9 ... # after try
```

On line 1, there is an instruction to enter into the exception handling. It specifies the label to go to when an exception is raised. Then, on line 3, the trap is popped, meaning we are no longer inside the try. Afterwards, a `goto` instruction prevents fallthrough to the trap handler. On lines 5 and 6, the code to start handling the trap occurs.

To turn this into WebAssembly, we turn back to the stackifier algorithm implemented previously. To do this, we add a special edge from the last block still within the try to the exception handler. This ensures that the exception handler will always appear after the try, thus allowing the WebAssembly exception handler to break out to it in the case of an exception. This means the `catch_all` block will not contain the exception handling code, but instead a branch to it. Then, before adding either blocks or loops, we insert a new `try` structure, starting with `push trap` and ending before `enter trap`. It is important that we end before `enter trap` rather than after `pop trap` since it is possible for there to be multiple `pop trap` statements per trap (although only one will ever be executed per trap). An observant reader may ask if this could end up with code after a `pop trap` being incorrectly included with a WebAssembly `try` block. Fortunately, the only thing that can immediately follow a `pop trap` is a `goto` statement.

3.6 Runtime

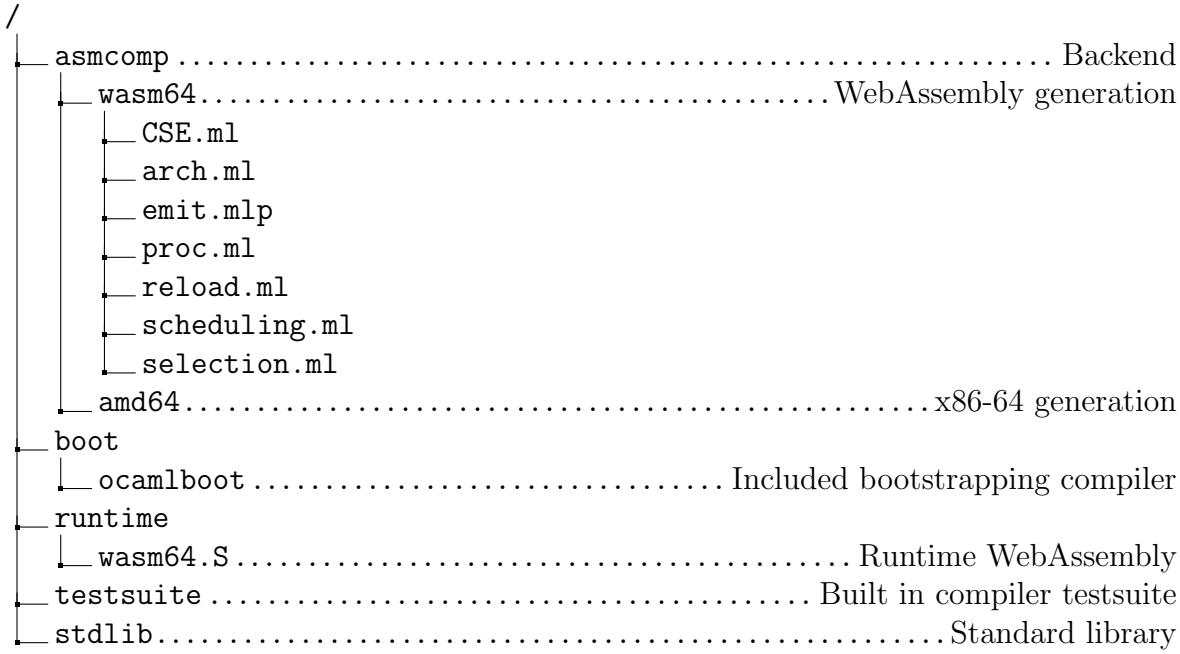
The final thing to do in order to create a working OCaml to WebAssembly compiler is to implement the runtime. As most of this is in C, all we need to do is write a small amount of startup code. This is where we shall define the globals mentioned earlier. We do all of this in a file called `wasm64.S`

The main thing to do here is to set up the globals. For example, in the following code, we copy the value of the young pointer from the state to the global (`caml_young_pointer`) we define to hold it.

```
global.get __caml_state
i64.const    Caml_state(young_ptr)
i64.add
i64.load 0
global.set __caml_young_ptr
```

We also define a function, `caml_call_gc`, which sets up the state, allowing for the GC to see our (likely modified) young pointer. It also puts the fake return address and stack pointer into the state so that the GC code can find the start of the stack frame. It then calls the GC proper. This function is called when we wish to perform a garbage collection from OCaml.

3.7 Repository overview



Chapter 4

Evaluation

We now begin the evaluation of the produced WebAssembly to OCaml compiler. This shall focus on two areas, correctness and performance. The former shall be evaluated using the compiler's test suite. The latter will focus on benchmarking the produced WebAssembly code.

4.1 Tests

The OCaml compiler ships with a test suite. This suite has hundreds of tests, testing the correctness of things such as floating point operations, garbage collection, etc. The tests work by compiling an OCaml program and verifying the output against a known good output. Unfortunately, it is not as simple as just building and then running the test suite. This is because running the compiled program is handled by a C program. This C program sets up the environment variables and manipulates file descriptors to capture the standard output and error of the executable. It then performs fork-exec to run the executable. This does not work on WebAssembly as Emscripten only provides stub implementations for the required syscalls (`dup2`, `fork`, `execvp`).

To solve this problem, we rewrite the C code to require only the `system` function. This is a simple function. It takes in a string as an argument and executes the string as though it were typed into the system's default shell. The function returns the exit code of the process it spawned. We are able to use this function, as the JavaScript runtime provided by Emscripten contains an implementation for `system` when running under Node.js. This is limited to Node.js as, for obvious security reasons, JavaScript running on the web cannot access the system shell. This does not pose a problem as we will only run the test suite under Node.js.

However, we are still faced with the problem of how to replicate the required functionality using only the `system` function. First, we shall consider setting the standard I/O file descriptors. The test suite tries to redirect these to files on disk. As such we can use standard shell redirections. For example, to write the output of a command to a file called `stdout.txt` and the error to `stderr.txt`, we execute `{command} 1> stdout.txt 2> stderr.txt`. If we wanted to append to instead of overwrite the file, we would swap `>` for `>>`. For setting the environment variables, we do not

provide an implementation. This is because the JavaScript runtime does not pass its environment variables through to WebAssembly. Environment variables are not used by the vast majority of the tests.

The tests provided are given in groups. The first set of tests we run is the `basic` test set. As the name suggests, this tests basic functionality. This includes features such as equality testing, integers, floats, arrays, pattern matching, function evaluation order and tail calls. All 39[1] tests passed. This is a strong indication that basic operations, local control flow and functions work correctly.

The `basic-more` tests were also run and all 20 passed successfully.

The next tests run were the `backtrace` tests. These tests were important to pass as they would confirm that the virtual return addresses are implemented correctly. This is because the backtraces are generated by walking the stack and looking these addresses up to find which line of source code generated them. Unfortunately, running these tests requires environment variables. This is to tell the OCaml program to print out the full backtrace on an unhandled exception. To run these tests, the JavaScript runtime was manually edited to include the required environment variable. After doing this, all tests passed successfully. This is very strong evidence that the virtual addresses work correctly and is a good sign that garbage collection will work. This test also verified that exceptions worked correctly.

To test garbage collection, we can run the `gc-roots` test. This is a very important test to pass as it would indicate that root collection for the GC works correctly (and by extension the GC itself). Fortunately, the test passed.

As some of the tests rely on functionality that we do not support in WebAssembly, such as dynamic linking[2], we cannot simply run all tests. However, the `misc`, `basic-float`, `basic-io`, `float-unboxing` and `lib-printf` all passed. In fact, no test tried failed due to incorrect code generation[3].

As a final test of correctness, it was verified that the produced compiler is self-hosting. This was done by compiling the compiler with itself, producing `ocamlopt.opt`. This WebAssembly-compiled compiler was tested on some trivial sample OCaml code and functioned correctly. In addition, all of the tests run by the test suite verified the output of `ocamlopt.opt` against `ocamlopt`. This is strong evidence that the compiler functions correctly as, if there were a bug, it would likely make itself evident in a complex program like the compiler.

Overall, the passing of many of the compiler's test suite tests and successful demonstration of the compiler's ability to self-host are strong evidence that the WebAssembly backend functions correctly.

[1] The `ocamltest` program was modified only to run tests on the native compiler.

[2] While WebAssembly itself can support this, to use this in OCaml would require implementing code to be able to read the WebAssembly binary format, something that is outside the scope of this project.

[3] Some tests failed due to missing C functionality in Emscripten.

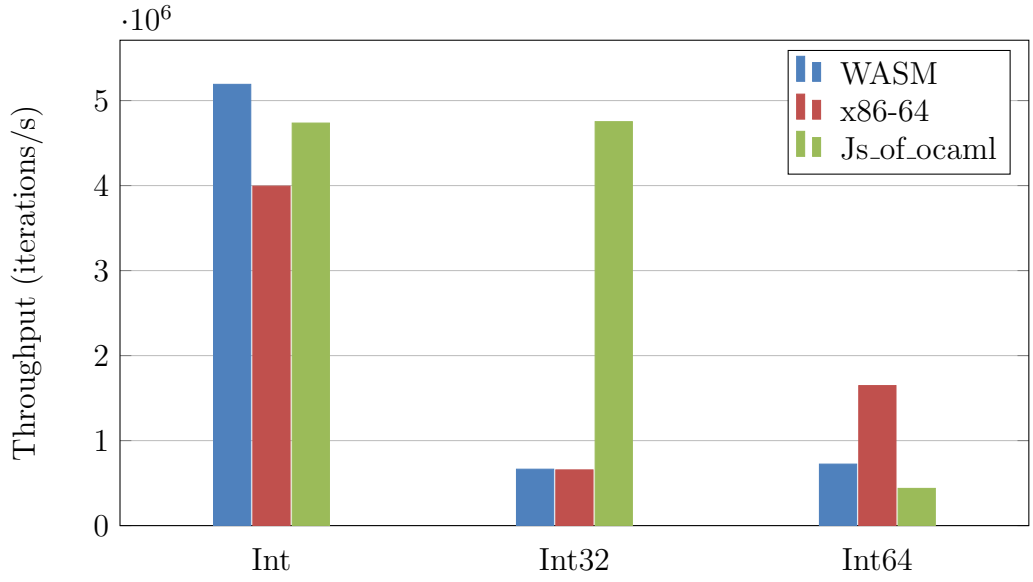


Figure 4.1: Integer performance

4.2 Performance

To benchmark performance, the benchmark[4] library was used. The generated WebAssembly was compared against a native x86-64 build using the OCaml 4.14.0 compiler and a Js_of_ocaml version 5.0.1 build. Both the WebAssembly and Js_of_ocaml build were executed using Node.js v19.9.0. All benchmarks were performed on an x86-64 machine with an Intel Core i9-13900K CPU running `Linux 6.2.9-arch1-1`[5].

First, the performance of integer operations was tested. This was done using the `numbers.ml` benchmark included with the benchmarking library. This test measures the throughput of addition on `Int`, `Int32` and `Int64`. It does this by spinning in a loop, incrementing an integer. This can be seen in the following code:

```
let f_int64 n =
  let rec loop i sum =
    if i < n then loop (i + 1) (Int64.add sum Int64.one) else sum in
  Int64.to_int (loop 0 Int64.zero)
```

The `f_int64` function is called in a loop until 10 seconds have passed. This is repeated 5 times for a total of 50 seconds for each of the integer types. Once this has been done, the average throughput and margin of error is calculated. The margin of error is not shown in any of the figures as in most cases it was under 1% with the most extreme being around 2%.

The results of the `numbers.ml` benchmark can be seen in Figure 4.1. It is surprising that WebAssembly and Js_of_ocaml beat out the native build. However, this is likely due to the 'turbofan' JIT compiler in the v8 engine used by Node.js undertaking profile-guided optimisations.

[4] https://github.com/Chris00/ocaml-benchmark/
[5] Obtained using `uname -sr`

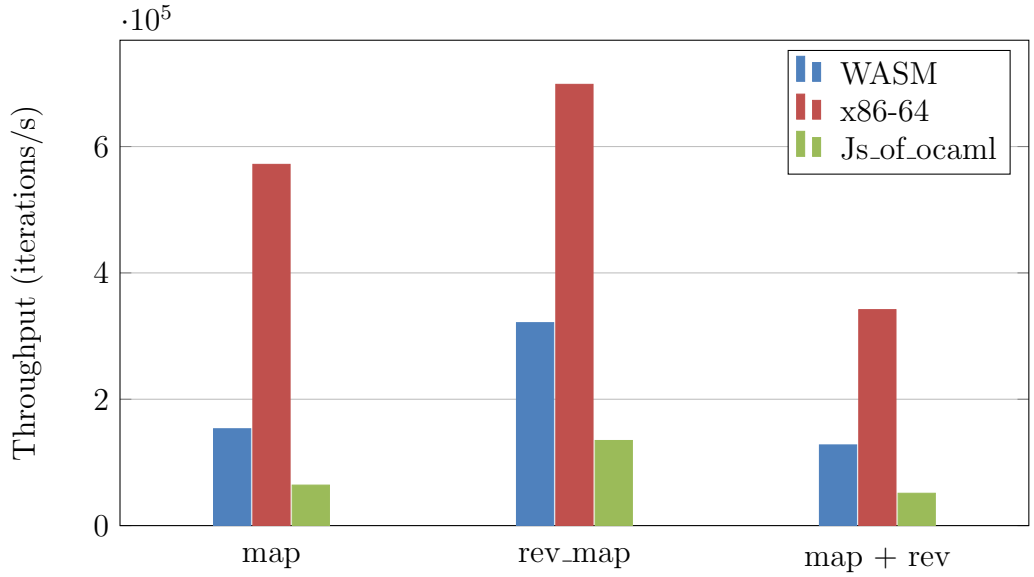


Figure 4.2: Allocation performance

Another thing to note is that, in general, the `Int32` and `Int64` types perform substantially worse than the default integer type. This is due to the fact that `Int32` and `Int64` are boxed[6] types. This does not, however, immediately explain why the performance hit is so substantial. In investigating this further, the first thing that was suspected was the GC. This was ruled out through simple profiling. By using perf[7], it was determined that the slowdown was caused due to cache misses.

A final thing to note is that Js_of_ocaml performs much better in `Int32` than WebAssembly and x86-64. This is because both `Int` and `Int32` are implemented as unboxed types on JavaScript. With the exception of this unusual case, WebAssembly beat out JavaScript.

The next test performed was designed to measure allocation performance. Three functions were written, the first using the `List.map` function to map a list over the `Fun.id` function; the second doing the same, but using the tail-recursive `List.rev_map` function instead; and the final using the `List.map` function followed by `List.rev`, computing the same as `List.rev_map` but in a non-tail-recursive way.

The results of this can be seen in Figure 4.2. WebAssembly was over twice as fast as Js_of_ocaml for all three tests. This is very likely due to the overhead of allocation and the GC in JavaScript. All three of WebAssembly, x86-64 and Js_of_ocaml showed significant performance increases in the `rev_map` test. This difference was more pronounced on WebAssembly and Javascript as opposed to x86-64. This is likely due to the extra overhead of function calls on these platforms compared to x86-64. Overall, this is a great result that shows how much faster WebAssembly can be than JavaScript.

The third benchmark to be undertaken is to test the performance of exceptions. This will be done using the `try_if.ml` test in the benchmarking library. The test benchmarks the performance impact of an out-of-bounds exception against the cost of bounds checking.

[6] A boxed integer is one that resides on the heap.
[7] A Linux tool to analyse performance.

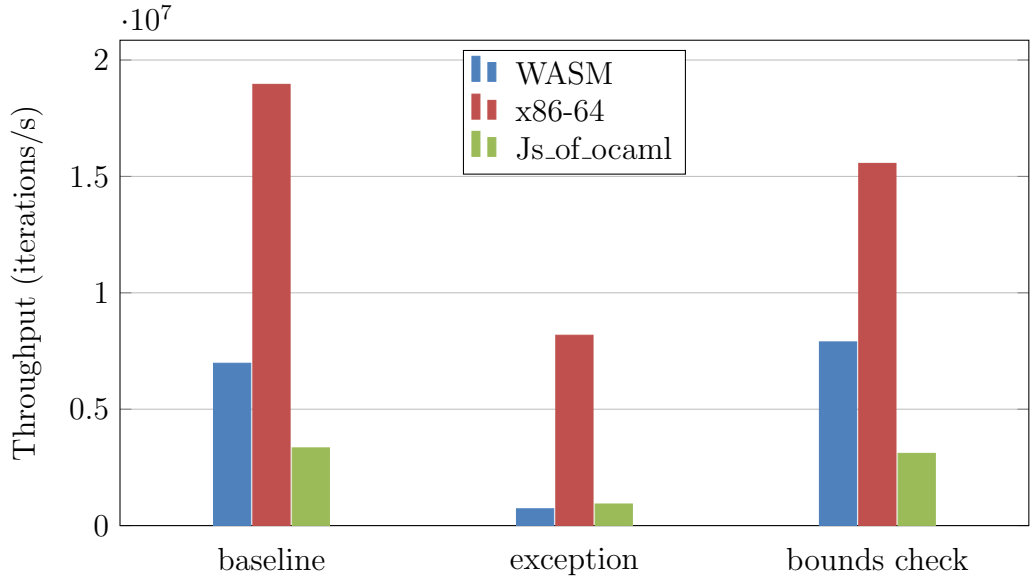


Figure 4.3: Out of bounds cost

This test is performed on an array with 100 elements. The baseline is obtained by reading each element from the array. The out-of-bounds and bounds-checking tests attempt to access all elements from the first to the 101st, thus, in the out-of-bounds case, triggering an out-of-bounds exception.

The results of the test can be seen in Figure 4.3. In keeping with the previous tests, WebAssembly significantly outperforms JavaScript. Sadly, exceptions pose an exception to this. It is also worth noting that x86-64 outperforms both WebAssembly and Javascript by, just shy of, a staggering nine times. This is likely due to OCaml's exceptionally fast method of implementing exceptions as an address stored in a register that can be jumped to at any time.

Since, for 100 of the 101 of the iterations, an exception is not raised, and since WebAssembly beat out Js_of_ocaml in the baseline test but lost in the exceptions test, we cannot be sure if the difference observed is significant. In order to test this, a final benchmark was written with an exception raised on every iteration of the loop. This can be seen in the following code:

```
exception Num of int

let [@inline never]ret n = n
let [@inline never]ret_exn n = raise (Num n)

let baseline_test a =
  ignore(List.fold_left (fun acc x -> acc + ret x) 0 a)

let exception_test a =
  ignore(List.fold_left
           (fun acc x ->
              acc + try ret_exn x with Num n -> n) 0 a)
```

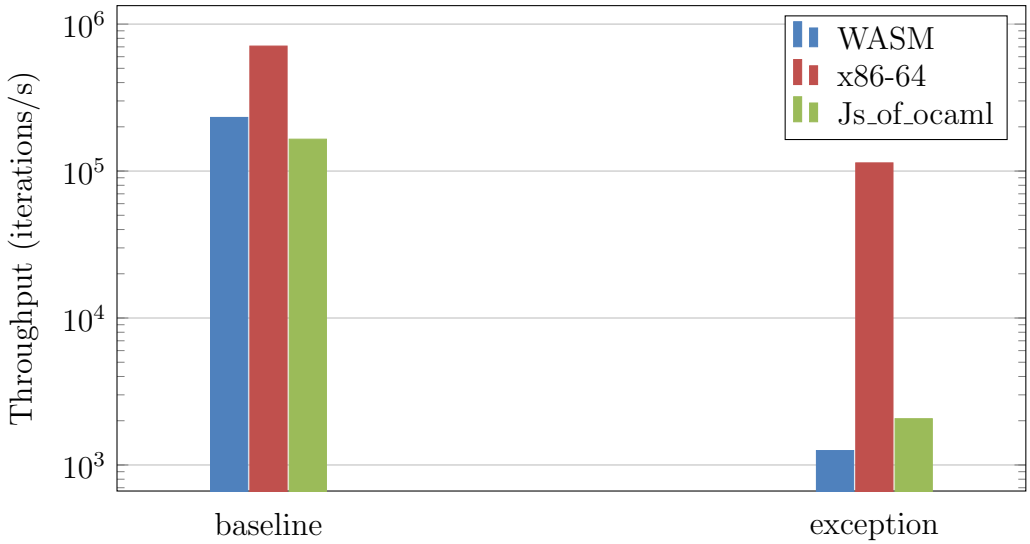


Figure 4.4: Exception

Looking at the benchmark results in Figure 4.4 and noting that this time the y-axis is logarithmic, we can see that WebAssembly and Js_of_ocaml perform about two orders of magnitudes worse than x86-64. While this is not surprising, sadly Js_of_ocaml outperforms WebAssembly by about 60%. This is likely due to either better mapping of OCaml exceptions onto JavaScript exceptions than is done for WebAssembly, or lack of optimisation of WebAssembly exceptions in the v8 engine used by Node.js due to their experimental nature.

Overall, the benchmarks performed here show that, with one exception, WebAssembly consistently outperforms JavaScript. This is especially true when it comes to allocation and garbage collection.

4.3 Decision against upstreamimg

The final thing to do with this project was to consider if it would be worth the effort to upstream it. Ultimately, this was decided against. There were several reasons for this decision.

The first and largest is due to the stage at which the backend was implemented. For most backends, implementing them at the usual Linear stage is the correct thing to do. However, due to the lack of unstructured control flow in WebAssembly, if this project were to be redone, it would probably be a better idea to implement the backend pre-linearisation. This would remove the need to implement the relooper algorithm and would prevent any future optimisations creating irreducible control flow from posing a problem. It would also make sense for the backend to be implemented before register allocation. The benefits of this would be two-fold. First, it would remove the need for a hacky system of mapping locals to virtual registers. Second, it would speed up compilation times, as register allocation rivals typing for making up the plurality of the compilation time.

A second reason why upstreaming was decided against is that the text representation

used is undocumented and not intended to be used outside of Emscripten. As such there is no documentation and it could change at any time. If this were to be redone, it would be a good idea to create an internal type to represent WebAssembly. This type could then be processed after it's created to locate symbols. This would also allow for direct WebAssembly binary generation, bypassing the need to write the WebAssembly as text to a file and then run an assembler. Not emitting text directly and using a custom type is somewhat similar to what the amd64 backend does within its `x86_dsl.ml` and `x86_ast.ml` files.

The final reason not to upstream is performance. While in most cases, a significant performance increase was seen over Js_of_ocaml, the performance of exceptions would need to be investigated before an upstream attempt were made.

4.4 Summary

Overall, the correctness of the compiler has been demonstrated with high confidence and some significant performance gains have been achieved over Js_of_ocaml. However, significant portions of the backend would need to be rewritten and improvements to the performance of exceptions would be needed before something like this could be shipped upstream.

Chapter 5

Conclusions

It has been shown that writing a WebAssembly to OCaml backend is not only possible, but is achievable with significant performance gains over alternative solutions. The backend was shown to function for the entirety of OCaml, allowing for any OCaml project to be compiled into WebAssembly.

5.1 Lessons Learned

Throughout the project, several lessons were learned. The first is the importance of doing something the right way rather than the simplest way. This is exemplified in the choice to generate WebAssembly from the Linear IR and the choice to generate WebAssembly in Emscripten's proprietary format. However, it is doubtful that doing these things differently would have been possible in the time allocated for the project.

Another thing learned is the complexity of real-world compilers. Real-world compilers have significantly more stages and perform a greater number of optimisations than could be taught in an undergraduate university course.

A final lesson learned is the value of being able to read code written by other people. Due to the lack of documentation on the internals of the OCaml compiler, a significant amount of time was dedicated to reading the source code of the compiler.

5.2 Future Work

Some reasonable future work would be to try to improve performance all around. This would include improving the performance of the WebAssembly generation, something on which very little time was spent for this project.

Further future work would be to implement the suggested changes in section 4.3. This would be a very significant undertaking. It would also be worth considering implementing the OCaml compiler as an LLVM frontend. This would allow for the targeting of many different architectures, as well as providing an opportunity for further optimisation of existing backends such as x86-64, something on which OCaml has a long way to go on to get up to the standards of compilers like GCC. This would also likely require OCaml to use

LLVM's garbage collection functionality, rather than relying on its own. For this reason and others, this would be an even larger undertaking than the suggestions in section 4.3.

Bibliography

[1] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9:366–371, 1966.

[2] BuckleScript. Difference from js_of_ocaml. `https://github.com/rescript-lang/rescript-compiler/blob/00ad78cbcfd1132d3a5931fe760706de35e480f6/site/docsource/Differences-from-js_of_ocaml.adoc`. [Accessed 10-May-2023].

[3] Thomas H. Cormen. *Introduction to Algorithms.*, volume 2nd ed, page 549. MIT Press, 2001.

[4] David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, jul 1980.

[5] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal on Computing*, 1(2):188–202, 1972.

[6] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 150–156, New York, NY, USA, 2002. Association for Computing Machinery.

[7] Yuri Iozzelli. Solving the structured control flow problem once and for all. `https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2`, 2019. [Accessed 07-May-2023].

[8] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, dec 1974.

[9] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Commun. ACM*, 16(8):503–512, aug 1973.

[10] Lyle Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4):893–920, oct 1988.

[11] Sander Spies. A webassembly backend for ocaml. `https://medium.com/@sanderspies/a-webassembly-backend-for-ocaml-b78e7eeea9d5`, 2018. [Accessed 09-May-2023].

[12] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.